Improving Large-scale Storage System Performance via Topology-aware and Balanced Data Placement

Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, Sudharshan Vazhkudai

National Center for Computational Sciences
Oak Ridge National Laboratory
Email: {fwang2,oralhs,guptas1,tiwari}@ornl.gov

***Abstract*—With the advent of big data, the I/O subsystems of large-scale compute clusters are becoming a center of focus, with more applications putting greater demands on end-to-end I/O performance. These subsystems are often complex in design. They comprise of multiple hardware and software layers to cope with the increasing capacity, capability and scalability requirements of data intensive applications. The sharing nature of storage resources and the intrinsic interactions across these layers make it to realize user-level, end-to-end performance gains a great challenge.**

We propose a topology-aware resource load balancing strategy to improve per-application I/O performance. We demonstrate the effectiveness of our algorithm on an extreme-scale compute cluster, Titan, at the Oak Ridge Leadership Computing Facility (OLCF). Our experiments with both synthetic benchmarks and a real-world application show that, even under congestion, our proposed algorithm can improve large-scale application I/O performance significantly, resulting in both the reduction of application run times and higher resolution simulation runs.

***Keywords*—Storage Area Network, Parallel File System, High Performance Computing, Performance Evaluation**

I. Introduction

Domain scientists often rely on high performance computing to simulate and understand physical phenomena, and to discover scientific knowledge in a wide range of disciplines. Large-scale scientific simulations can be both compute and I/O intensive as they produce and process large amounts of data in a very short period of time, which stresses and stretches the capability of file and storage system. This *bursty* I/O pattern may arise out of the applications' own practical needs, or, it can be an artifact of so-called defensive programming, where periodic checkpoints are used to compensate for possible failures of large-scale compute clusters. With the advent of big data and the new crop of data-intensive applications, the amount of data being generated is likely to increase significantly, hence, performance and capabilities of file and storage systems will become even more critical.

Unfortunately, a file and storage system with a designed peak throughput does not always lead to higher performance at the application level due to multiple factors. First, an I/O subsystem is typically shared among multiple consumers with different usage patterns, leading to potential load imbalances and increased contention. Second, the architecture of I/O subsystems is becoming increasingly complex. For example, an I/O request may travel through multiple paths and may experience different delays at different components it traverses. At Oak Ridge Leadership Computing Facility (OLCF), we observe firsthand that such a complex I/O subsystem suffers from severe contention and that there exists a significant load imbalance among different components in the storage system, as shown in Section III.

To address the I/O load imbalance and contention issues, we propose a topology-aware, balanced placement strategy that is based on a site-defined, tunable, weighted cost function of selectable resources. The strategy is topology-aware because it assigns weight factors to separate resource components depending upon which storage layer they belong to, as different storage layers have different degree of impact on the end-to-end performance. It is balanced as it keeps track of usage of all storage components and balances the load along the end-to-end I/O path.

For evaluation purposes, we have implemented our algorithm as an easy-to-use, user-space library and performed extensive experiments on the Titan supercomputer [1] and Lustre based Spider II file and storage system [2]. A detailed discussion on Titan's end-to-end I/O path and Spider II can be found in Section II.

By repeating small-, medium-, and large-scale experiments over extended period of time in a production environment with different compute node allocation layouts on Titan, we demonstrate that the proposed technique is effective in improving application I/O performance at different scales. More importantly, it is not dependent on any fortunate node allocation layout. Further, we integrate our library with a large-scale scientific application, S3D. Our evaluation indicates that the proposed scheme provides significant improvements to S3D application even in a noisy, production environment on Titan supercomputer.

It should be noted that, our proposed strategy can also be implemented as system-wide resource load-balancer. However, for simplicity and without loss of validity, we decide to start out as a user-space library because it is easy to test, evaluate and deploy such prototype as compared to a system-wide library on a production environment. It is conceivable and desirable to integrate our proposed algorithm with popular I/O middleware solutions such as HDF5 [3], [4], [5] and ADIOS [6], [7], [8] so that applications can effortlessly take advantage of the performance benefits without having to interface with yet another library and modify their I/O code.

Although the implementation and experimental context of our work are centered around Titan and Lustre, the load imbalance issue in large-scale storage systems is not uncommon at other large-scale compute clusters as indirection layers are being employed to scale out the infrastructure. Given the popularity of the Lustre parallel file system [9], we think that our path of exploration and proposed methods may find wider applicability, and should be beneficial to the cluster computing community at large. Also, it is expected that the amount of data being generated from the scientific simulations will continue to increase in the big data era. Therefore, we believe our proposed strategy to improve the I/O performance of large-scale data-intensive scientific applications is likely to become even more important in the near future.

The contributions of this paper are three fold. First, we empirically show that how a simple yet typical I/O use case in a large-scale, layered file and storage system can lead to I/O load and resource use imbalance. Second, we propose and implement a topology-aware, balanced placement strategy to address the load and resource use imbalance issues. Third, we demonstrate, with both synthetic benchmarks *and* a real-world scientific application, that this strategy can indeed mitigate the problem and improve application I/O performance significantly regardless of the layout of compute node allocation. It is the latter point that takes this approach beyond Titan-specific environment and makes it generally applicable to other compute and storage infrastructures as well.

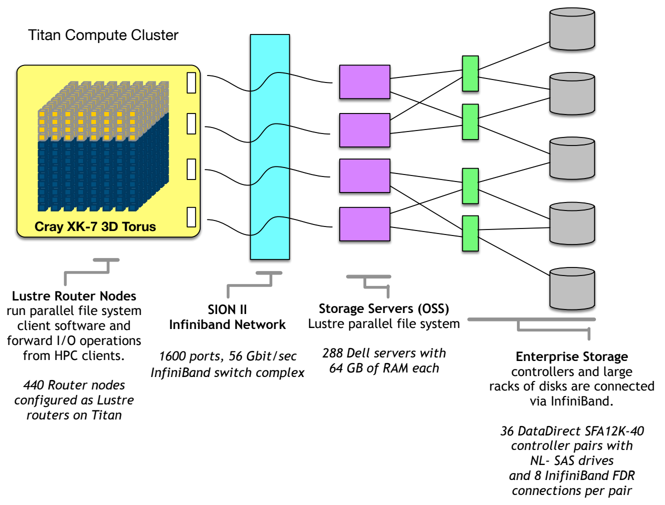
The rest of paper is organized as follows. In Section II, we provide a detailed description of the Titan and Spider II infrastructure. Its complex I/O path and empirically observed congestion points motivate the design of the end-to-end, balanced placement strategy, which is elaborated in Section III. In Section IV, we discuss our experimental setup, evaluation strategy and testing results. In Section V, we present an example of application integration, our experience and results. Finally, Section VI summarizes the paper and discusses the future work.

II. Background and Related Work

Since we evaluate our proposed technique on the Titan supercomputer and Spider II parallel file system, this section aims at presenting relevant architecture details of Titan and Spider II. Also, since the I/O traffic of compute clients will traverse interconnect network in both directions (read and write), the placement of I/O routers have a tremendous impact on traffic pattern. The past and current research efforts on interconnect routing congestion avoidance indirectly motivate our proposed solution. Therefore, we will review these and other resource balancing related works in the latter part of the section.

Titan is a Cray XK7 system with 18,688 compute nodes, 710 TB of total system memory [10]. This high capability compute machine is backed by a center-wide parallel file system known as Spider II. Spider II is one of the world's fastest and largest POSIX complaint parallel file systems. It is designed to serve write-heavy I/O workloads generated by Titan compute clients and other OLCF resources. The topology and architecture details of Spider II infrastructure are illustrated in Figure 1 and described as follows:

On the back-end storage side, Spider II has 20,160 disks organized in RAID 6 arrays. Each of these RAID arrays act as a Lustre Object Storage Target (OST). An OST is the target device where the Lustre parallel file system does file I/O (read or write) at the object layer. These OSTs are connected to Lustre Object Storage Servers (OSSes) over



Titan Compute Cluster
Cray XK-7 3D Torus
Lustre Router Nodes
run parallel file system client software and forward I/O operations from HPC clients.
440 Router nodes configured as Lustre routers on Titan
SION II
Infiniband Network
1600 ports, 56 Gbit/sec InfiniBand switch complex
Storage Servers (OSS)
Lustre parallel file system
288 Dell servers with 64 GB of RAM each
Enterprise Storage
controllers and large racks of disks are connected via InfiniBand.
36 DataDirect SFA12K-40 controller pairs with NL- SAS drives and 8 InifiniBand FDR connections per pair

Fig. 1: Infrastructure and I/O path between Titan (Cray XK-7) and its backend storage

direct InfiniBand FDR links; these OSSes currently run Lustre parallel file system version 2.4.2. There are a total of 288 OSSes and each OSS has 7 OSTs (a total of 2,016 OSTS). Spider II is configured and deployed as two independent and non-overlapping file systems to increase reliability, availability, and overall meta data performance. Each file system, therefore, has 144 Lustre OSSes and 1,008 Lustre OSTs.

Each OSS is connected to a 36-port IB FDR Top Of the Rack (TOR) switch. Each TOR switch is connected with a total of 8 OSSes. Each switch also connects to two 108-port aggregation switches. The aggregation switches provide connectivity for the Lustre meta data and management servers.

On the front-end at the compute side, there are two different types of nodes in Titan: compute and Lustre I/O router nodes. Both types of nodes are part of the Gemini network [11] in 3D torus topology. Each node has a unique network ID (NID) for addressing purposes. A total of 440 nodes on Titan are configured as Lustre I/O routers, 432 of which are used for routing Lustre file I/O traffic between Titan and Spider and 8 are used for routing Lustre meta data I/O. Titan I/O routers are connected to SION TOR switches via InfiniBand FDR links. Note that the SION TOR switches enable these I/O routers to reach to the back-end storage system (OSSes and OSTs).

In order to provide connectivity over different networks and communicate between file system clients and servers over these networks, Lustre provides a network abstraction layer called LNET (Lustre Networking) [12].

Lustre can route traffic between multiple networks of the same or different types. This functionality is provided by Lustre I/O routers. By default, Lustre uses a round-robin algorithm to pick routers. The first alive router on top of the list will be picked to route the message and then will be placed at the end of the list for the next round to provide a load balance among multiple routers. Each SION TOR switch is assigned an LNET route. Our proposed technique focuses on balancing the load on resources from the Luster I/O routers up to the OSTs. The algorithm improves I/O performance by intelligently selecting I/O end points (Lustre router, LNET, OSS and OSTs).

Since the interconnect networks in HPC environments are the backbone for both message exchange and I/O traffic, optimizing network communications has been a popular research topic. Fine grain routing in the context of Lustre file system [13] targets at minimizing congestion in the system-wide 3D-torus interconnection network. A preemptive resource throttling approach was introduced in [14]. This approach presented an open loop end-point throttling of number of messages in flight per core or throttling number of cores per node to increase performance scalability. Another congestion control mechanism was presented in [15]. The proposed method was again based on throttling resources and specifically designed for HPC clusters with InfiniBand networks. The design concept was to limit and load-control the multipath expansion in order to maintain low and bounded network latency for I/O traffic. A follow-up work was presented in [16]. Predictive and Distributed Routing Balancing (PR-DRB) technique was proposed to monitor messages latencies on I/O routers and record solutions to congestion, to quickly respond in future similar situations. A Step-Back-on-Blocking (SBB) flow-control mechanism that primarily addresses the allocation effectiveness in high-radix interconnection networks was proposed in [17]. This method combined the advantages of the wormhole and cut-through routing algorithms for torus networks, while adding a means for adaptive allocation of the communicational resources. The proposed SBB mechanism was shown to provide low message latency and achieved high fraction of the maximal channel bandwidth by performing conditional evasion of temporary blocked network resources or traffic hot-spots.

HPC network communications, such as I/O, is a special case since that most of the traffic is short-lived and bursty. Adaptive routing mechanisms are shown to be highly effective for long-lived traffic [18], however, they have not been deeply evaluated for HPC environments. To the best of our knowledge and experience, it is the specialized HPC networking hardware's responsibility to optimize the network and I/O traffic. The Quadrics network [19], is a good example to that, which uses a network operating system and specialized hardware to support high-performance data transfers for HPC environments. Of these HPC I/O specialized solutions, perhaps one of the most mature ones is PaScal [20]. It had several interesting technologies, including multi-level switch-fabric interconnection network, bandwidth on demand. As mentioned earlier, our focus in this work is to provide further performance improvements by enabling applications to direct I/O intelligently across layers along the I/O path.

I/O load imbalance is a known problem for the HPC domain and data centers [21]. Multiple approaches have been proposed to solve this problem. In [22], it was proposed to modify the PVFS file system [23] to achieve better I/O load balancing. In [24], authors described a load imbalance problem for cloud data centers. Their algorithm was designed to adjust the two end points on the I/O path, computational virtual machines and virtual disks, to balance the overall load in a data center. Other approaches have been proposed such as, replicating data or moving the I/O intensive compute jobs to eliminate hot spots [25]. To the best of our knowledge, there has not been a proposed solution aimed at solving the end-to-end I/O load imbalance problem within the constraints of a large-scale HPC production environment.

Another thread of research tackling the general resource allocation and optimization problem is to use the generic knapsack algorithms [26], [27], [28], [29], [30]. In particular, the precedence constrained knapsack problem for HPC domain has been studied by [31], [32]. However, these approaches focus only at job scheduling to optimize compute system utilization, and have not proven to be applicable to the I/O load imbalance problem in the HPC domain. The next section describes our balanced placement strategy.

III. Balanced Placement Strategy

In this section, we describe our placement algorithm that aims to balance per job I/O resource allocations. In the most general case, the problem can be formulated as:

$$C = w_1R_1 + w_2R_2 + w_3R_3 \ldots + w_nR_n,$$

subject to $w_1 + w_2 + w_3 + \ldots w_n = 1$, where C is the cost of an I/O path being evaluated, R_i is the resource component along the I/O path and w_i is the weight factor assigned to the resources. If the goal is to minimize the I/O cost, then a weight factor reflects the likelihood of the particular type of resources to be a point of contention. Resources in the case of our evaluation can be logical I/O routes (i.e., LNET), or actual file system and networking devices (i.e., Lustre I/O routers, OSSes, OSTs, and SION InfiniBand TOR leaf switches). We aim at distributing the I/O traffic evenly across resource components to avoid point of contention. However, as to be discussed in the following section, such a scheme needs take into consideration of the topology and resource dependencies.

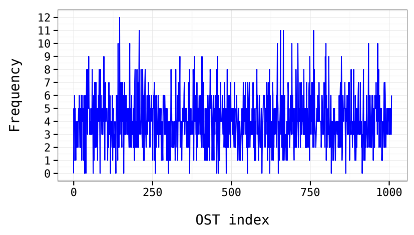
A. *Need for Balanced Resource Usage*

To understand the need for balanced placement and justify why the proposed algorithm works, we conduct the following illustrative experiment.

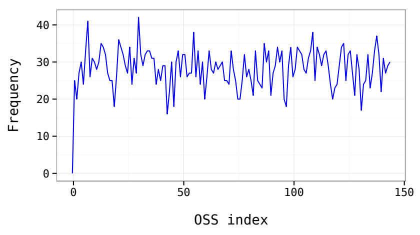
We launch 4096 processes with each process doing a single file I/O operation against half of the Spider II file system. The traces of those files are analyzed to examine the utilization distribution of different components.

Figure 2 (a), (b) and (c) shows the resource usage distribution for OSTs, OSSes and LNETs, respectively. Recall that there are a total of 1008 OSTs, 144 OSSes, and 18 LNETS in one half of the Spider II file system. We observe that there exists a significant variation in usage across across components of any given type (e.g., OST, OSS or LNET). For example, some OSTs are used more than 10 times while some others are never used (corresponding to zero frequency count). Similarly, OSSes and LNETs show significant usage-imbalance under the default placement strategy. Consequently, imbalanced resource utilization increases the contention at certain components more than others.

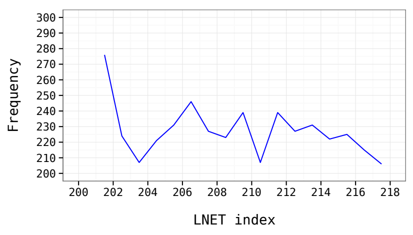
One may think that a naive sequential round-robin allocation may fix this. However, we show that round-robin allocation scheme doesn't resolve this issue. We illustrate this by using an example that only involves OSTs (for simplicity)



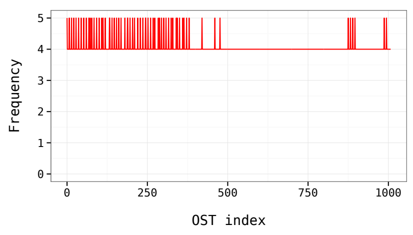
(a) Default OST Placement



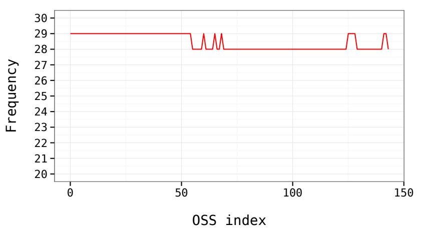
(b) Default OSS Placement



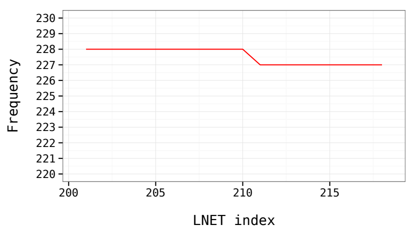
(c) Default LNET Placement



(d) Balanced OST Placement



(e) Balanced OSS Placement



(f) Balanced LNET Placement

FIG. 2: COMPARISON OF RESOURCE USAGE DISTRIBUTION: DEFAULT VS. BALANCED

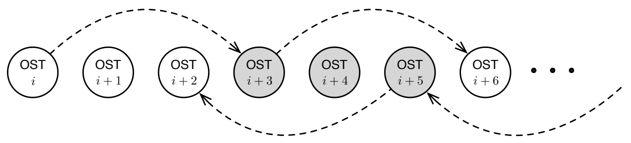


FIG. 3: OST ALLOCATION EXAMPLE

and not the components from other layers such as I/O routers and LNETS.

Figure 3 shows a set of six OSTs, three of them are assigned to one OSS and the rest three to other OSS. They are distinguished by different gray codes. After allocating OST_i, if we allocate the next target to be OST_{i+1}, then the two processes are going to compete for the resources within a single physical OSS (as these two OSTS reside within the same OSS). The dotted arrow line represents an allocation scheme that takes OSS boundary into consideration. As we scale up the number of I/O processes, we may have to eventually loop back and allocate more than one OST per OSS. The take-away of this example is to show that one needs to take resource usage of other components and their dependencies into account, and not solely focus on components in one layer in isolation. Focusing on components in one layer in isolation may still result in imbalanced resource utilization. Next, we present a detailed description of balanced placement algorithm.

B. Balanced Placement Algorithm

Recall from Section II that an I/O request on Titan traverses through a complex path and gets allocated multiple resources on the way. In particular, I/O requests coming out of a compute node goes to an I/O router node via high-speed Gemini interconnect. From I/O routers, these I/O requests traverse to SION leaf switches via "logical" I/O network (called LNET, from here on). Depending on its destination OST, there could be different logical LNETs an I/O request can cross. Next, from SION leaf switches these I/O requests go to Lustre OSSes, and then to Lustre OSTs.

A placement strategy can be simply viewed as an assignment/binding of an I/O client (be it a compute node or MPI process) to an OST. However, an I/O request can take multiple possible paths (via different I/O routers, LNETs, and OSSes) to reach the same OST. A strategy that attempts to place an I/O request in a balanced fashion across these resources (I/O router, LNET, OSS and OST) should have a way to quantify the cost of a particular I/O path. To this end, we define a placement cost function that takes weighted average of how frequently different resources have been used by previous I/O requests originating from the same application.

$$\begin{aligned}\text{placement cost} = w_1 \times \ \text{rtr\_freq} + w_2 \times \ \text{lnet\_freq} \\ + w_3 \times \ \text{oss\_freq} + w_4 \times \ \text{ost\_freq}\end{aligned}$$

where rtr_freq, lnet_freq, oss_freq and ost_freq are usage frequency of I/O routers, LNETs, OSSes and OSTs. Respective relative weight factors are denoted as w_1, w_2, w_3, and w_4. Given a compute node, the algorithm loops over all the reachable OSTs to choose one with the lowest placement cost (as shown below). The same process is repeated for all the compute nodes allotted to the application. Note that this function is invoked *only once* before an application enters I/O phase.

```
1: procedure BALANCED PLACEMENT (List of NIDs, List of OSTs)
```

```
2:
3:     lnet_freq ← 0, rtr_freq ← 0
4:     oss_freq ← 0, ost_freq ← 0
5:     random_offset ← randomly selected reachable lnets
6:     for all NIDs in the input NID list do
7:         lnet ← random_offset
8:         for all reachable OSTs do
9:             cost ← MAX
10:            oss ← ost2oss()                ▷ map OST to OSS
11:            mycost ← placement_cost(
12:                lnet_freq, rtr_freq ,oss_freq, ost_freq)
13:            if mycost < cost then
14:                mycost ← cost
15:                picked_ost ← ost
16:                picked_oss ← oss
17:            end if
18:        end for
19:        record NID and the selected OST
20:        increment lnet_freq, rtr_freq, oss_freq, ost_freq
21:    end for
22: end procedure
23:
```

Next, we discuss the key design issues and choices of our proposed algorithm. First, how do we decide the weight factors? Generally speaking, these tunable parameters are site dependent, which require a careful analysis and systematic evaluation to identify the possible congestion point. We take advantage of our decade-long experience with these file systems to assess which components are utilized in a relatively more imbalanced fashion. As a generic observation, load across OSTs are more imbalanced compared to other components. It led us to set the value of w_4 higher and evenly split the rest as: $w_1 = 0.2$, $w_2 = 0.2$, $w_3 = 0.2$, and $w_4 = 0.4$ in our prototypes. We do not claim that these weight factors are optimal. However, Figure 2 (d), (e) and (f) show that these weight factors resolve the load imbalance issue. Our evaluation results show that these heuristic settings can improve performance significantly. Further fine-tuning of these weight factors may remove a few outliers visible in Figure 2 (d), (e) and (f), however we believe these outliers may not result in additional significant performance gains than what our current weight factor based design already provides.

Second, what is the overhead of this strategy? Our scheme incurs modest computational overhead, because the algorithm is invoked only once before each I/O phase. The internal data structures ensure that storage overhead associated with each allocation requests are kept to minimal and resource can be freed as soon as possible.

Third, will invocations of the same algorithm from different applications create additional source of contention? We are careful in ensuring that it doesn't create contention among applications. We instantiate the LNET selection with a random offset ensuring that different application have different starting points and hence, less likely to contend for the same paths and OSTs. We note that our proposed algorithm would be complimentary to system-wide contention-aware resource allocation scheme, as it doesn't create any new artificial source of contention.

Finally, under what settings will this algorithm not perform well? Our algorithm is sensitive to the size of the possible resources (for example, reachable OSTs) and routing paths. When number of I/O processes are small *and* tightly packed in close proximity, they are likely to have access to a set of less optimal OSTs and hence, corresponding routes as well. In such cases, our algorithm has relatively less opportunity to perform load balancing. This is confirmed by our experiments to be discussed in details in the next section, where we achieve relatively modest performance improvements at low node counts.

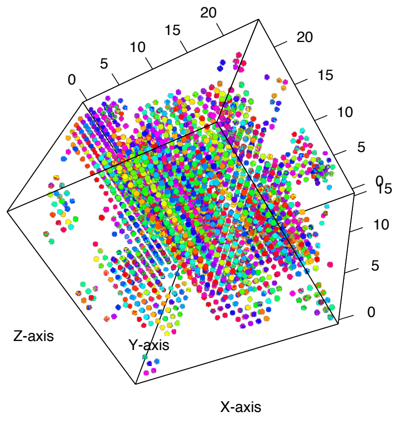


FIG. 4: COVERAGE OF NODES FOR OUR EXPERIMENTS ON TITAN: X AND Y-AXIS REPRESENT THE ROWS AND COLUMNS OF CABINETS, AND Z-AXIS IS A CABINET IN THE VERTICAL DIRECTION.

In the next section, we will discuss the experimental setup and evaluation results using this placement strategy.

IV. EXPERIMENTAL SETUP AND EVALUATION

In this section, we first describe the experimental setup, then we present and analyze our evaluation results both from a synthetic benchmarking tool and real scientific application.

We rely on synthetic benchmarking for assessing the strength and weakness of our approach because (1) it is not always possible to test with a varied range of parameters and perform sensitivity studies with real application codes; (2) compute allocation time is very expensive on Titan supercomputer: it is calculated that the operational cost is about \$1 per node per hour. Since Titan has 18,688 compute nodes, it amounts to an estimated \$18,688 per hour. Therefore, testing with a synthetic application that only stresses I/O provides significant cost savings.

Our synthetic benchmarking tool, referred to as Placement I/O (PIO), is specifically designed to do comparative analysis. We also discuss our experience of application integration by instrumenting a large scale scientific application, S3D [33], a high-fidelity turbulent reacting flow solver to demonstrate the effectiveness our proposed scheme on Titan supercomputer.

A. *Experimental Setup*

We perform all our experiments on Titan supercomputer. There are two major issues that we address to demonstrate that our results are representative.

First, we run all our experiments in a busy production environment. That is, we didn't take advantage of maintenance quiet period to perform our experiments. We ran all our

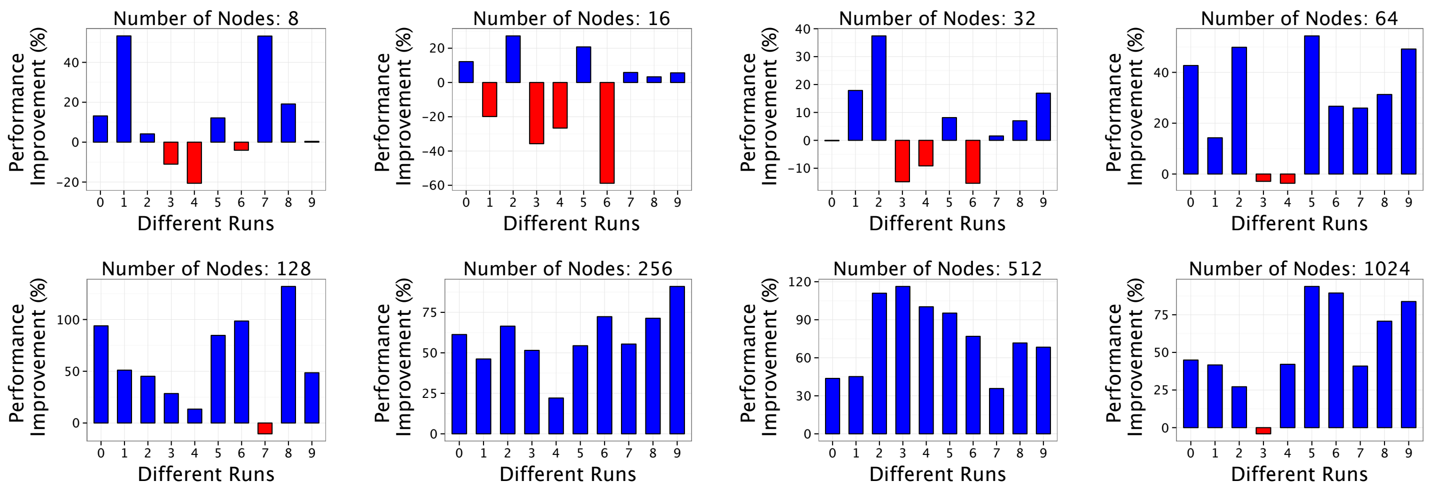


FIG. 5: PERFORMANCE IMPROVEMENT OVER DEFAULT PLACEMENT, PIO-BASED SCALING TESTS ON TITAN.

experiments while other users were running their scientific experiments as well. We believe that presenting such results shows that our performance gains can be and are observed in a *active production environment*.

Second, we show that our experiments cover a broad set of nodes on Titan, instead of a small subset of cherry-picked nodes. This demonstrates that the performance gains of our library can be achieved by any application irrespective of what set of nodes the scheduler may allocate. Next, we explain how our experiment design achieves a decent coverage of whole Titan supercomputer.

For a submitted job, application Level Placement Scheduler (ALPS) used on Titan tends to return the allocated node-list where nodes are logically close to each other as much as possible (to reduce communication latency variance). Therefore, there are two ways to achieve a higher node coverage over multiple runs: (1) submit scaled runs one after another through dependency specification such that each run will be scheduled independently and hopefully cover a different set of compute nodes; (2) submit scaled runs simultaneously, conjecturing that if some previously submitted large-scale job finishes and frees up many compute resources, our experiments can therefore occupy a larger area of coverage.

Since both approaches have merits, we mix and match our runs using both methods to gain broader coverage. In the end, we were able to obtain results from more than 30 scaled runs, with each run to sample allocation point ranging from as few as 4 nodes to as high as 1024 nodes. For each allocation point, we perform multiple iterations for both default placement and balanced placement, resulting more than 3000 data points. We keep track of the allocated nodes for each run, remove redundant nodes, and convert the logical NID into a 3D point in the Torus topology. The result is visualized by Figure 4. It shows that through repeated runs, we achieve a notably high coverage of overall allocation space.

B. Placement I/O Benchmarking Tool (PIO)

For evaluation purpose, we designed and implemented a new MPI-based placement I/O benchmarking tool, PIO. It

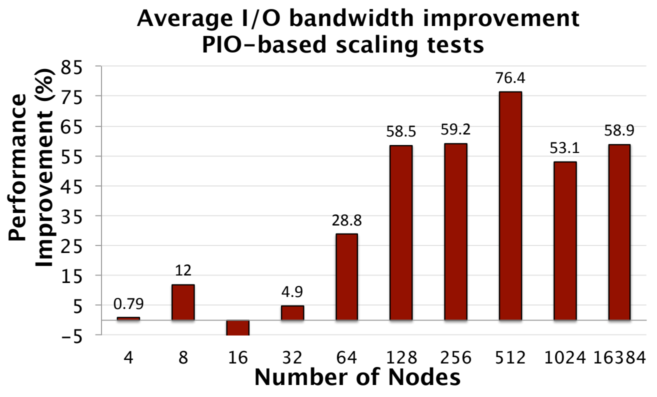


FIG. 6: AVG. PERFORMANCE GAINS (IN %): BALANCED PLACEMENT VS. DEFAULT PLACEMENT

operates in two modes, the default mode and the placement mode. For the default mode, it follows closely with what an IOR benchmark tool [34] does. We have empirically validated the results which are comparable given the same set of runtime parameters in POSIX I/O: block size, transfer size, fsync option. For the balanced placement mode, PIO make use of MPI call to gather the running compute node IDs, invoke the placement I/O library for data placement on OSTs, pre-create the files using Lustre's `llapi` library with specific layout, then perform the I/O operations. This allows us to conduct a side-by-side comparison given the same set of compute node allocation on Titan. In addition, PIO maintains and outputs detailed I/O timing information for each MPI process as well as allocation information for post-analysis.

C. Evaluation Results

The results of the scaling runs are summarized in Figure 5. Each sub-figure represents a particular node allocation, scaling from 8 nodes to 1024 nodes. The X-axis represents enumeration of runs of the same count of node allocation, but for *different* set of nodes.

Since our experiments are conducted in a noisy, active production environment, the absolute performance number gained during one run may not always be conclusive. To address this issue, we perform multiple runs over extended period of time and *at least* three iterations in each run. Note that iterations within one run get the same set of nodes for both default and balanced placement strategy; this is essential for fair comparison between these two schemes per iteration. While different runs are allocated on different set of nodes, enabling us to cover a broad set of compute nodes on Titan. In other words, this methodology helps us average out the variance across the same set of allocated nodes and also cover a large set of Titan compute nodes. We use arithmetic average of multiple iterations within same run for comparison. We use performance improvement in percentage as our metric for comparison: Performance Improvement $= 100 * (BW_{\text{balanced\_placement}}/BW_{\text{default\_placement}} - 1)$.

We present PIO-based scaling results in Figure 5; a negative value is colored in red and positive improvement is colored in blue for better visual representation in color. From the figure, we make following observations. First, the effectiveness of the our proposed scheme is relatively small or insignificant in some cases when running at small scale (up to 32 nodes). Second, as we scale up in terms of I/O processes, our proposed scheme consistently yields significant performance improvement. We even logged more than 100% performance improvement in multiple cases (see Figure 5 for 128 and 512 node runs. For example, in one run with 512 nodes, the *average* performance gain of three iterations is from 25.6 GB/s to 55.3 GB/s). Third, while there are variations across different runs, we observed that the trend remains the same and there have been consistent performance gains across multiple runs and iterations.

Next, we show the average performance improvement across runs in Figure 6. It confirms that for more than 128 I/O nodes, we consistently observe more than 50% speed up with balanced placement algorithm.

We also conducted experimental runs at close to the full scale of Titan, a total of 16,384 nodes participating the I/O. These results are not shown in Figure 5, since we have only two runs of results at this particular scale comparing to tens of runs at the smaller scale. Nevertheless, it clearly shows that balanced placement algorithm can improve per job I/O performance for more than 50% at this extreme scale as well.

D. Effect of the Stripe Count

The use of striping serves two major purposes: (1) provide high-bandwidth access to a single file, (2) improve performance when OSS bandwidth is exceeded. We note that our current implementation and evaluation assumes that the stripe count parameter is set to one. This is an practical limitation imposed by the Lustre, and not by our balanced placement algorithm per se: Lustre, as it is, supports a restricted form of stripe layout to the user land, i.e., the first starting index of OST only. There are ongoing development to lift this restriction, but this has not materialized yet.

Even with this restriction in place, we conducted experiments to show how increasing the stripe count to four (the default value as of Lustre version 2.4) in the base case will

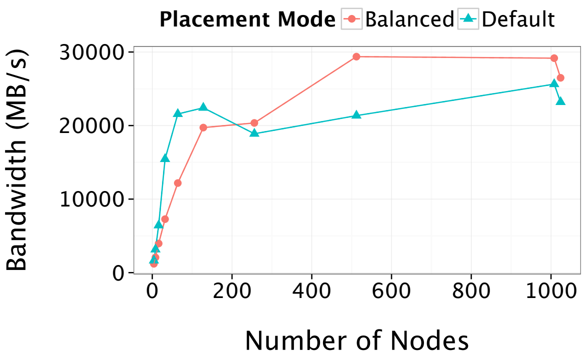


FIG. 7: BANDWIDTH COMPARISON: BALANCED PLACEMENT (STRIP COUNT = 1) VS. DEFAULT PLACEMENT (STRIP COUNT = 4)

compare with our proposed scheme. The experimental results, as shown in Figure 7, suggests that as the system scales, our proposed scheme with stripe count of one can actually outperform default I/O with default stripe count of four. We believe that when stripe layout with multiple OSTs is possible, our algorithm can take full advantage of it and deliver even better scaling results. We also note that increasing the stripe count is likely to increases the pressure on meta-data severs, and hence, it doesn't always yield performance over the stripe count equal to one. We show, in the next section, that S3D performs better with the stripe count equal to one compared to the stripe count equal to four, at large node counts. However, our proposed scheme continues to improve performance when applied with stripe count equal to one.

V. APPLICATION INTEGRATION

In this section, we demonstrate the effectiveness of our balanced placement approach by integrating placement library (libPIO) with a large-scale scientific application, S3D, a high-fidelity turbulent reacting flow solver [33]. S3D writes state of the simulation to the file system which is later used for analysis. The data file also serves as a checkpoint to restart. Both MPI collective I/O and Fortran I/O are supported in this application. We use Fortran I/O because the I/O bandwidth achieved by Fortran I/O has been shown to perform better than MPI I/O [35].

Only 30 lines of code [36] were needed to be added/modified in the checkpoint subroutine of the application to integrate the placement library. Primary additions were the `init()`, `nid2ost()` and `finalize()`. A `lfs setstripe` call was also to added to provide the desired placement of output files on Lustre OSTs. In the prototype version of the library, the list of node-ids (nids) is not stored in libPIO and hence each `nid2ost()` call is surrounded by a gather operation for nids from all the ranks, and scatter operation for ost-ids to all the ranks. In the future work, we plan to make the interface of libPIO more robust and the `init()` call itself will provide the nids to libPIO. In this manner each rank would be able to call nid2ost() by passing its nid to get a placement suggestion (i.e. OST) for the particular rank (obviating the need of gather and scatter operation for nids and OST-ids respectively). The libPIO is written in C and S3D code is primarily written in

Fortran, so we had to include some extra code as well to provide bindings and wrappers to library calls which is not needed if an application is writte in C/C++.

One key difference in the methodology of PIO benchmarking tool and S3D is that by default S3D uses all the cores present on a compute node (i.e. 16 cores per node on Titan). This packs more work on a compute node to improve the computational efficiency of the simulation. Please note that, each compute node on Titan runs a single OS and there is a single mount point per file system on a given compute node. From libPIO point of view, all 16 cores on a compute node share the same file system end point (i.e. Lustre OST). Therefore, S3D's approach creates additional pressure on a Lustre OST. Despite this additional pressure, our balanced placement scheme performs reasonably well as discussed next.

We perform scaled runs with 150, 375, 750, 1,125, 1,875 and 3750 nodes which corresponds to 2,400, 6,000, 12,000, 18,000, 30,000 and 60,000 MPI processes, respectively. We use weak scaling of problem grid size such that each process generates a 27 MB of output/checkpoint file periodically (11 such checkpoints in each run). The I/O bandwidth measurement is performed for default placement (both stripe count of 1 and 4) and balanced placement by running three S3D simulations in the same allocation back-to-back (same as PIO methodology).

In Figure 8, we present the summary of I/O bandwidth improvements observed for S3D from using placement library relative to using default placement. The improvements are averaged over ten runs for each configuration. We compare our balanced placement approach with system default stripe count of 4 as well as stripe count of 1. From Figure 8, we observe that smaller node count (i.e. 150, 375 and 750) runs show better performance with stripe count 4 than default or balanced approaches with stripe count of 1. As the node count grows, 1,125 nodes and larger, we observe that balanced placement approach can outperform the default approach with both stripe counts 1 and 4. This is consistent with performance results in Section IV-D from synthetic benchmark experiments. For node count of 1,875 and 3,750, there is significant improvement in I/O bandwidth when libPIO is used with S3D for balanced I/O placement.

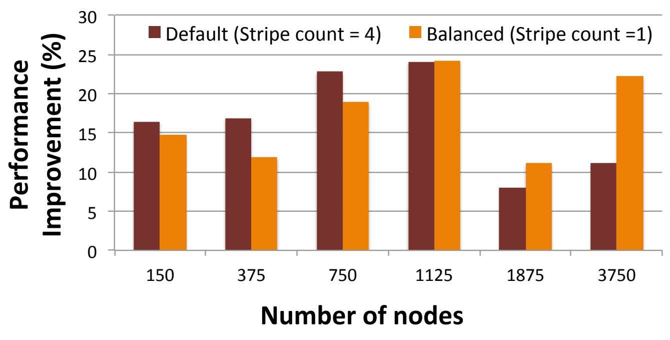


FIG. 8: AVERAGE I/O BANDWIDTH IMPROVEMENT FOR S3D (IN %): BALANCED VS. DEFAULT PLACEMENT (RELATIVE TO DEFAULT PLACEMENT, STRIPE COUNT = 1)

To summarize, the performance of libPIO is lower when node/processor count is smaller, which we also observed in the case of PIO benchmark (Figure 7). For large node/processor counts, our experiments conclude that libPIO for scientific applications, like S3D, shows very promising results. Although our tests were conducted in a production (noisy) environment, we observed substantial gains in I/O performance. Also, the ease of implementation with minimal code changes makes us believe that libPIO can be widely adopted by scientific users and middleware I/O libraries.

VI. DISCUSSION AND CONCLUSIONS

While center-wide shared file systems, like the one deployed at Oak Ridge Leadership Computing Facility, have their own advantages, it remains challenging to translate the raw capability into visible and predictable improvement for user applications. This is due to a variety of reasons, ranging from the shared design of the system and the contention caused by complex nature of storage infrastructure and highly sophisticated and often non-deterministic end-to-end I/O routing paths.

This paper proposed a topology-aware, end-to-end I/O resource allocation strategy, which aims at distributing the load on I/O paths in a more balanced fashion. This strategy is implemented as a user-level, easy-to-use library, and evaluated extensively, at-scale on the Titan supercomputer and Spider II file and storage system. We demonstrated that I/O performance can be improved by more than 50% on a per-job basis using synthetic benchmarks. Our experiments with a real-world scientific application showed that our balanced placement algorithm is effective in improving write I/O bandwidth even in a noisy (i.e., active production) environment. Moreover, the simplicity of integrating the library into a real scientific application gives us the confidence that it is a viable solution for scientists to utilize for improving their applications' I/O performance.

Although our evaluation platform is centered around Titan and Spider II, the load and resource imbalance issue are quite common in large-scale storage infrastructure. This is especially true as more indirection layers are employed to drive capability, capacity and scalability to the next level and to meet the demands of big data. We believe our path of exploration and proposed techniques can find wider applicability in the future and benefit community at large.

ACKNOWLEDGMENT

We thank Bradley Settlemyr and Douglas Fuller for their feedbacks on earlier drafts of this paper, and Gary Liu for sharing his experience on ADIOS integration.

This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

References

[1] A. S. Bland, J. C. Wells, O. E. Messer, O. R. Hernandez, and J. H. Rogers, "Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory," in *Proceedings of Cray User Group Conference (CUG 2012)*, May 2012.

[2] S. Oral, D. A. Dillow, D. Fuller, J. Hill, D. Leverman, S. S. Vazhkudai, F. Wang, Y. Kim, J. Rogers, and J. Simmons, "OLCF's 1 TB/s, next-generation lustre file system," in *Cray User Group*, Napa Valley, California, May 2013.

[3] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 2011, pp. 36–47.

[4] M. Folk, A. Cheng, and K. Yates, "HDF5: A file format and i/o library for high performance computing applications," in *Proceedings of Supercomputing*, 1999.

[5] M. Howison, "Tuning HDF5 for Lustre file systems," in *Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, 2010.

[6] "ADIOS: the adaptive I/O system," https://www.olcf.ornl.gov/center-projects/adios/.

[7] J. Lofstead, S. Klasky, S. K., N. Podhorszki, and C. Jin, "Flexible IO and Integration for Scientific Codes Through The Adaptable IO System (ADIOS)," in *CLADE 2008 at HPDC*. Boston, Massachusetts: ACM, June 2008. [Online]. Available: http://www.adiosapi.org/uploads/clade110-lofstead.pdf

[8] G. Liu, "Personal communications," 2014.

[9] Sam Bigger, "Why use Lustre," https://wiki.hpdd.intel.com/display/PUB/Why+Use+Lustre, 2011.

[10] J. Dongarra, H. Meuer, and E. Strohmaier, "Top500 supercomputing sites," http://www.top500.org, 2009.

[11] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini system interconnect," in *Proceedings of IEEE High Performance Interconnects (HOTI)*, August 2010.

[12] Intel Inc., "Lustre File System, Operations Manual for Lustre - Version 2.0," http://wiki.lustre.org/images/3/35/821-2076-10.pdf, 2011.

[13] D. A. Dillow, G. M. Shipman, S. Oral, and Z. Zhang, "I/O congestion avoidance via routing and object placement," in *Proceedings of Cray User Group Conference (CUG 2011)*, 2011.

[14] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu, "Congestion avoidance on manycore high performance computing systems," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304594

[15] D. Lugones, D. Franco, and E. Luque Fadón, "Dynamic routing balancing on infiniband network," *Journal of Computer Science & Technology*, vol. 8, 2008.

[16] C. N. Castillo, D. Lugones, D. Franco, and E. Luque, "Predictive and distributed routing balancing for high speed interconnection networks," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 2011, pp. 552–556.

[17] P. Borovska and D. Kimovski, "Adaptive flow control in high-performance interconnection networks," *The Journal of Supercomputing*, pp. 1–24, 2013.

[18] P. Geoffray and T. Hoefler, "Adaptive routing strategies for modern high performance networks," *High-Performance Interconnects, Symposium on*, pp. 165–172, 2008.

[19] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics Network: High-Performance Clustering Technology," *IEEE Micro*, vol. 22, pp. 46–57, January 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=623303.624502

[20] G. Grider, H. Chen, J. Nunez, S. Poole, R. Wacha, P. Fields, R. Martinez, P. Martinez, S. Khalsa, A. Matthews, and G. Gibson, "PaScal - a new parallel and scalable server IO networking infrastructure for supporting global storage/file systems in large-size Linux clusters," *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, p. 46, 2006.

[21] J. D. Rosario and A. Choudhary, "High performance I/O for massively parallel computers: Problems and prospects," *IEEE Computer*, vol. 27, no. 3, pp. 59–68, 1994.

[22] Y. Zhu, H. Jiang, X. Qin, D. Feng, and D. Swanson, "Improved read performance in a cost-effective, fault-tolerant parallel virtual file system (CEFT-PVFS)," in *Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid*, 2003, pp. 730 – 735.

[23] R. B. Ross, R. Thakur *et al.*, "Pvfs: A parallel file system for linux clusters," in *in Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, pp. 391–430.

[24] A. Singh, M. Korupolu, and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 53:1–53:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413424

[25] X. Qin, H. Jiang, Y. Zhu, and D. Swanson, "Dynamic load balancing for i/o-intensive tasks on heterogeneous clusters," in *High Performance Computing - HiPC 2003*, ser. Lecture Notes in Computer Science, T. Pinkston and V. Prasanna, Eds., vol. 2913. Springer Berlin Heidelberg, 2003, pp. 300–309. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24596-4_32

[26] E. Horowitz and S. Sahni, "Computing partitions with applications to the knapsack problem," *J. ACM*, vol. 21, no. 2, pp. 277–292, Apr. 1974. [Online]. Available: http://doi.acm.org/10.1145/321812.321823

[27] L. Caccetta and A. Kulanoot, "Computational aspects of hard knapsack problems," *Nonlinear Analysis: Theory, Methods & Applications*, vol. 47, no. 8, pp. 5547 – 5558, 2001, proceedings of the Third World Congress of Nonlinear Analysts. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0362546X01006587

[28] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[29] M. S. Hung and J. C. Fisk, "A heuristic routine for solving large loading problems," *Naval Research Logistics Quarterly*, vol. 26, pp. 643 – 650, 1979.

[30] C. Chekuri and S. Khanna, "A ptas for the multiple knapsack problem," in *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '00. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000, pp. 213–222. [Online]. Available: http://dl.acm.org/citation.cfm?id=338219.338254

[31] C. Chekuri and R. Motwani, "Precedence constrained scheduling to minimize sum of weighted completion times on a single machine," *Discrete Applied Mathematics*, vol. 98, no. 1âĂŞ2, pp. 29 – 38, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166218X98001437

[32] S. G. Kolliopoulos and G. Steiner, "Partially-ordered knapsack and applications to scheduling," in *Algorithms - ESA 2002*, ser. Lecture Notes in Computer Science, R. Mohring and R. Raman, Eds., vol. 2461. Springer Berlin Heidelberg, 2002, pp. 612–624. [Online]. Available: http://dx.doi.org/10.1007/3-540-45749-6_54

[33] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond," in *Proceedings of the International conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 2012, p. 15.

[34] "IOR HPC benchmark tool," http://sourceforge.net/projects/ior-sio.

[35] W. Liao, A. Ching, K. Coloma, A. Nisar, A. Choudhary, J. Chen, R. Sankaran, and S. Klasky, "Using mpi file caching to improve parallel write performance for large-scale scientific applications," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 2007, pp. 1–11.

[36] S. Gupta, "Patches for S3D using libPIO API," https://gist.github.com/saurabg/11259267, 2014.